**Using reverse engineering techniques to see how a common malware packer works**

April 12, 2016

**UPX (Ultimate Packer for eXecutables) is an open source executable packer that is common in the malware scene (albeit often heavily modified). UPX supports all major operating systems and both x86 and x64 platforms. UPX on its own features no anti-debug checks, no scrambled code/stolen bytes and no encryption. For this post I have coded my own software in the C language to demonstrate how UPX works, what it does to the .code/.data segment in the PE header and how you can rebuild an executable that has been packed with UPX.cram**

Packers like UPX are used to scramble and mask in an effort to make it more difficult for the analyst/reverser to figure out what is going on. Within the malware scene, UPX is used as an “outer” layer. Malware writers use UPX and a secondary, often a custom made packer that is not detected by AV software. This stops unpacking scripts from working, instead the user has to manually unpack the file. Reversers who run their target through a detection tool will only see the outer layer, in our case UPX along with the version it was packed with. Some packers are blacklisted by AV software, like Themida. These are automatically detected as malware due to the signature of the packer being blacklisted. UPX on the other hand is not blacklisted, which is why UPX is so popular. UPX works by compressing the sections stored within the Section Table of the PE file. A strong indicator of UPX being used is the renaming of the header names (UPX0/UPX1). The main purpose of UPX is to reduce file size, this helps mask the malware as a .jpg or to spread through emails.

**How do l determine if a program is packed manually?**

I present four very simple guidelines to keep in mind when researching your executable.

**#1**

* What functions are being used?
* Is the application large, but only uses a few imports?
* LoadLibaryA and GetProcAddress are two functions that are used widely by packers.

**#2**

* String table is missing or contains only junk; a missing, corrupted or encrypted string table is usually a pointer that a PE packer has been used. Some packers (like UPX) like to add their own strings.

**#3**

We will see large amounts of “data” inside the executable, this means that the debuggers analysis module is displaying the junk code (encrypted functions from the packer) as “data”.

Its unpacked or packed code that haven’t been analysed yet, otherwise it would have been shown in clear form. (See reference picture for clearance).

**#4**

Section names are not using standard names (UPX0/UPX1/SR for Armadillo) instead of .data/.code.

Some of the more advanced packers utilize an army of features to make the life of the reverser hell. These include, but are not limited to, on the fly encryption/decryption to prevent inline patching and CRC checks to prevent tampering. Malware developers also utilize several techniques to make it harder for you to dump the executable.

Some of the techniques used are:

**SizeOfImage**

Malware developers change the value of the SizeOfImage variable stored within the PEB (Process Environment Block). This prevents debuggers to be attached to the process and prevents memory dumpers from working correctly.

**Section Header**

Tampering with the Section Header prevents commonly used software from gathering information about the image sections, this prevents dumping of the Image Address Space.

**Self-Debugging**

Self-debugging is used to prevent another debugger from being attached to the parent process. It relies on the simple fact that only one process can be debugged at one time in User-Mode. Self-Debugging works by creating a process, which then attaches to itself by acting as a debugger. Armadillo uses this technique with its nanomite feature.

**Exceptions**

Using exceptions can make your disassembler/debugger do all kinds of fun stuff, from crashing, suddenly exiting to running in a loop like a crazy hamster. The usage of exceptions can also make the reversing process much harder, as an example “Spotify” uses exception handling to crash several of the popular disassemblers.

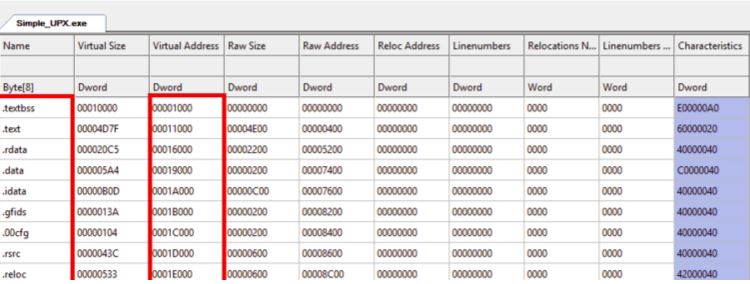
Most Anti-Debugging techniques work in conjunction with Anti-Packing techniques.

The purpose of dumping an executable is the following:

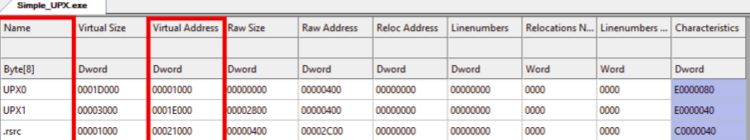
Re-create the executable, in its original form, before it was packed.

* This allows us to perform static analysis on the now unpacked “payload” data.
* Removal of unwanted data sections.
* Increased clarity, no obfuscating to make our progress harder.

Here is what the Section Headers look like in our unobfuscated program.

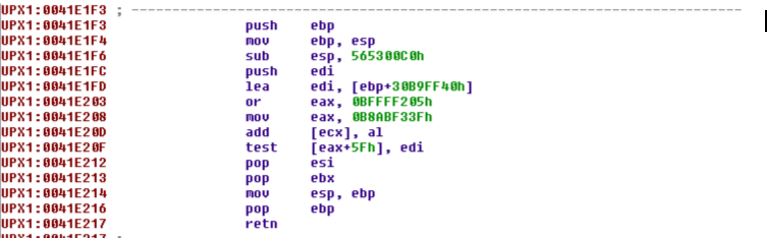


As we can see, both the section names and the VA is intact. Now we look at what happens when we apply UPX.

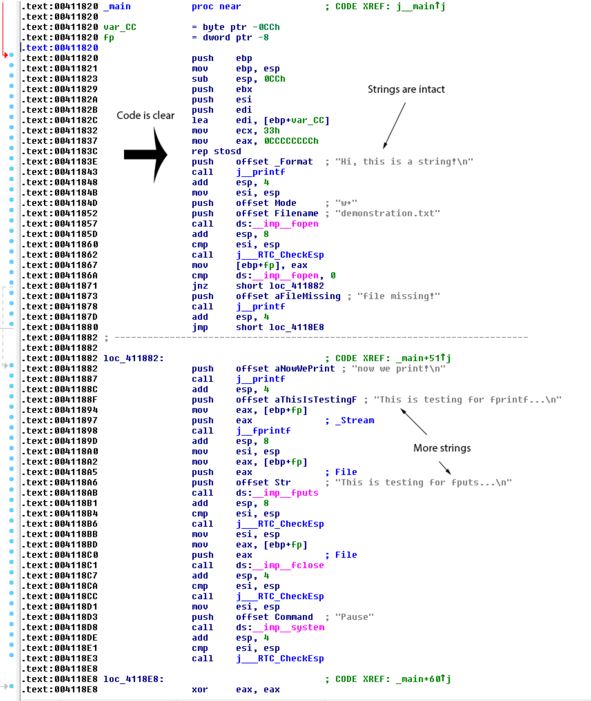


UPX relocates the sections then renames them. It then alters the entry point where we will run a short routine to jump to OEP. UPX1 contains the stub code and this code will unpack the real program code that lies in UPX0.

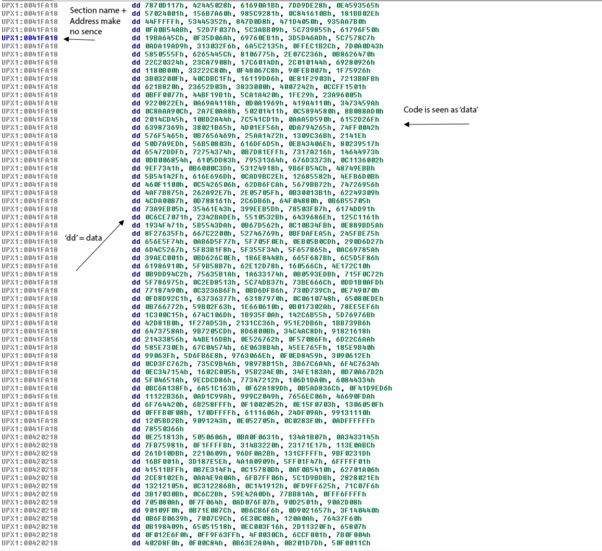
This is the routine for UPX unpacking;



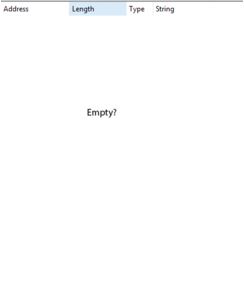
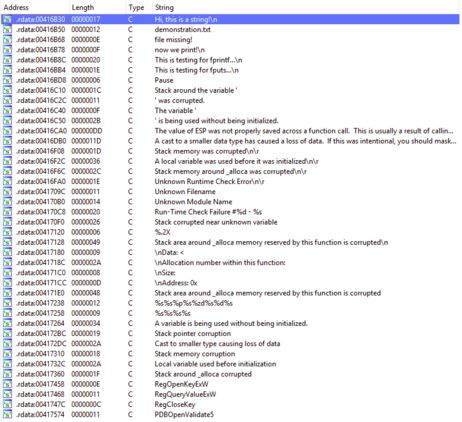
How we see the code unobfuscated.



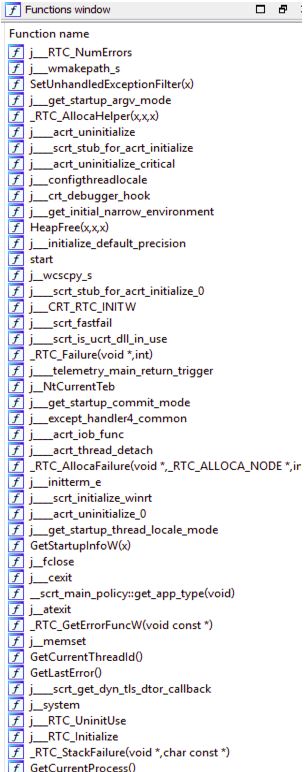
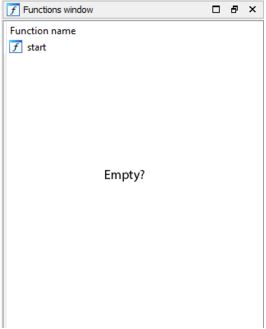
How we see the same code:



The string windows show us everything unpacked and nothing when packed.



Packed vs Original



**Source code for the program I coded**

#include <stdio.h>

#include <stdlib.h>

int main()

{

printf("Hi, this is a string!\n");

FILE \*fp;

fp = fopen("demonstration.txt", "w+");

if (fopen == NULL)

{

printf("file missing!");

}

else

{

printf("now we print!\n");

fprintf(fp, "This is testing for fprintf...\n");

fputs("This is testing for fputs...\n", fp);

fclose(fp);

system("Pause");

}

return 0;

}

Remember;  
*“If it runs, we can unpack it.”*